

## Overview

- motivation; Q& Browser case study
- modelling concurrency with UML
- parallelization methodologies
  - Parallel Application Design Layers (PADL)
    - ◆ software architecture, concurrency models (parallel design patterns), application frameworks (UML), APIs
  - Predicate Breakdown Structure (PBS) and logical fault tolerance
- intermission
- defect removal (debugging) methodologies
- parallel program correctness
  - defects in OpenMP programs: race conditions, non-thread-safe functions, deadlocks
  - tools for threaded program correctness
- Assignment 1: questions / discussion

(lecture largely based on Ch 6–10 of *Professional Multicore Programming* by Hughes & Hughes, now on Google Books)

## Motivation

- huge range of applications that can benefit from parallelism:  
e.g. teleconferencing tools, compilers, software analysis tools, simulators, virtual environments, entertainment
- no single methodology or tool (silver bullet) in order to do this
- extra challenges of parallelism even more strongly require systematic approaches

### Case Study: Lexicon Agent-based System

- consider a system to identify unrecognized words from a text file
  - potentially great degree of parallelism!
- a *main agent* creates *word expert agents*
- use a *dictionary lexicon* to recognize known words
- if not known, place into an *unrecognized words list*
- *misspelled* and *morphology search agents* to filter this list

## Concurrent UML

- UML is a standard tool for design and modelling (covers OO, agent-oriented and event driven paradigms)
- motivation: essential for developers to fully understand system at design stage
- provides techniques for visualizing a system from the structural, behavioral and architectural perspectives
- e.g. Dictionary lexicon class structure: attributes and methods (e.g. [Hughes&Hughes, Fig 9.2(b)])
- (concurrent) method behaviors can be classed as **isQuery**, **sequential** (callers must ensure mutual exclusion, i.e. acquire a lock), **guarded** (object itself ensures mutual exclusion) or **concurrent** (integrity inherently guaranteed with concurrent access)
  - e.g. `addWord(word: string): boolean (guarded)`  
`nextWord(void): string (isQuery, concurrent)`
- ref: Sparx System UML Tutorials and White Papers

## Concurrent UML (II)

- overall structure is provided using dependency stereotypes and associations  
e.g. Lexicon system [Hughes&Hughes, Fig 9.10]
- active objects may be classified as a **process** or a **thread** ([Hughes&Hughes, Fig 9.12])
- collaboration diagrams shows control flow behavior between objects ([Hughes&Hughes, Fig 9.13])
- time ordering of (major) interactions is indicated by sequence diagrams ([Hughes&Hughes, Fig 9-14])  
these indicate when objects are active or inactive
- finally activity diagrams show flow of control from one (atomic) action or (decomposable) activity ([Hughes&Hughes, Fig 9-16])  
swim-lanes indicate activities for a particular object, with synchronization bars indicating concurrent fork/joins

## Parallel Application Design Layers

a 5-layer analysis model for parallel application design and implementation

1. software architecture: provides the infrastructure and supports the basic functionalities
  - multiagent and blackboard
2. concurrency models (and also parallel design patterns)
3. application framework: identify class hierarchies, component libraries and languages to be used
4. APIs: expression of system in code, e.g. OpenMP, Posix threads
5. (operating) system: interactions including system calls, scheduling of threads, etc (e.g. for low-level debugging or performance analysis)

We will be mainly concerned with the first three stages (covering the design phase).

## Multiagent Architectures

- a flexible model, used to capture goal-oriented work patterns
  - in turn leads to goal-based decompositions and state transitions
  - can help develop declarative formulations for parallel programming
- uses two or more agents that can be concurrently active (abstracted over process & threads)
- before proceeding to Level 4, relationships and interactions of all agent need to be refined
- case study: Guess Code
  - guess the correct 6-character alphanumeric code; code may be changed as time runs out, requiring more speed (by more search agents)!
  - activity diagram for Agent-based formulation [Hughes&Hughes, Fig 8-2]

## Blackboard Architecture

- (ref: wiki) basically two types of components:
  1. blackboard: centralized object containing current problem state, constraints, goals, event queues; used for co-ordination; may be segmented for concurrent (read and/or write: CREW or CRCW) update
  2. problem solvers: software with specialized knowledge in the problem domain
    - sometimes formulated in terms of condition-action pairs; ideally are highly autonomous
- ideal for problems which can be broken into semi-independent sub-problems
- typical memory configurations ([Hughes&Hughes, Fig 8-3])
- control may be in blackboard, problem solvers or an external agent
- case study: domain-specific Q&A Browser (plain-text questions)
  - problem solvers: parse questions, determine word morphology, find mis-spelt words, analyse semantics, analyse pragmatics
  - these in turn update the blackboard with their partial results (Lexicon system is a part of this)

## Concurrency Models

- shared data access model can be characterized as CRCW, CREW or EREW (Concurrent/Exclusive Read/Write)
- task model may be characterized as SIMD or MIMD
  - reflects data decomposition and functional & data decomposition respectively
- select concurrency model appropriate for the application (and architecture)
- boss-worker: boss creates threads, places work unit in queue and wakes up workers; worker takes next work unit from queue (suspend if not available)
- peer-to-peer: single agent creates all threads; then all equally execute work (creating new work units if need be)
- producer-consumer: producer places work in shared queue, consumer removes
- pipeline: multiple stage version of above; multiple work units processed simult.
- example: [Hughes&Hughes, Fig 8.5, Tab 8-4];

	architecture	conc. model	memory model	SIMD/MIMD
Q&A Browser	blackboard	peer-to-peer	CREW or EREW	MIMD
Guess Code	multiagent	boss-worker	EREW	SIMD

## Parallel Design Patterns: Refinement of Concurrency Models

- refs: Resource page; motivation:
  - isolate the (||) communication aspects of an application into well-known scenarios
  - systematic 'cookbook', common language, code re-use
- stems from functional and data decompositions of application determined by inherent data and control dependencies
- main patterns: slides by Eun-Gyu Kim '04, slides 8-14  
Embarrassingly Parallel, Replicable, Repository, Divide-and-conquer, Pipeline, Geometric
- question: which (PADL) concurrency model and architectures are suitable for each of these?
- case study: Parallelizing BZip2, Pankriatus et al.2008: many possibilities
  - splits file into blocks (Embarrassingly Parallel)
  - compress block: BWT, MTF and Huffman stages (Pipeline)
  - MTF stage (Replicable - bucket sort)

## Predicate Breakdown Structure

- PBS can capture statements about the rules, constraints and assertions that apply between parts of the system (objects, agents, etc)
- captures the (intended) meaning of the system, not its operation
- Guess Code example could have the following PBS:  
P1 you've won the game if your guess is correct and in time  
P2 the guess is correct if it is valid and corresponds to the last code given  
P3 the guess is in time if it is within the time limit  
P4  $N$  agents are enough to find the code within the time limit
- should be formulated with level 5 of PADL (analysis stage)
- can be built into the system in the form of checks for errors (e.g. P2 violated) or 'impossible worlds' (e.g. P4 violated)
- exception handlers (e.g. C++/Java) can be used to catch and deal with these errors (logical fault tolerance)
- such declarative methodologies can help with defect survival as well as removal

## Defect Removal (Debugging)

- testing can reveal various defects:
  - race conditions, control flow errors, resource exhaustion, non thread-safe code, deadlocks ; calculation errors and other sequential defects
- challenge: reproduce conditions for the 1st 5 of these for debugging
- methodology: does the design model and PBS correctly capture the solution?  
does the implementation model correctly implement the design model?  
are all concurrency issues properly treated?
- OpenMP debugging methodology:
  - verify sequential version: drop `-xopenmp` flag, guard OpenMP function definitions with `#ifdef _OPENMP`; reverse order of parallel loops
  - verify parallel version:
    - ◆ find minimum number of threads (1?) exposing bug
      - ★ bugs requiring large number of threads likely to be from data races
    - ◆ run with minimum optimization (may expose bug with `flush` directive)
    - ◆ selectively disable/enable directives to locate the faulty section
    - ◆ check all (library) functions called in parallel regions are thread-safe

## Defects in OpenMP Programs: Data Race Conditions

- loops without independent iterations are subject to data races, e.g.

```
#pragma omp parallel for
for (i=0; i<n; i++)
    a[i] = a[i+1] + b[i];
```
- when different threads can access same array element; (also `j` should be `private`)

```
#pragma omp parallel for
for (i=0; i<n; i++) // need make j-loop a critical section
    for (j=0; j<3; j++) // different threads could use same a[]
        a[m[i]] += p[j] * q[i];
```
- compiler may re-order assignment, or value of `first` may propagate ahead

```
static int first = 1;
static double a; // likely to be initialized to 0
double inv() {
    if (first) a = 10.0, first = 0;
    return (1.0 / a); // 2nd thread may get here before a==10.0
}
...
#pragma omp parallel
{ int i = omp_get_thread_num();
  x[i] = inv(); }
```

## OpenMP Defects: use of Non Thread-Safe Function and Deadlock

- non-thread-safe function: global data not protected by concurrent writes:

```
static int count = 0;
int foo() {
    count++; // may miss increments!
    return g(count);
}
...
#pragma omp parallel for private(i)
for (i=0; i<N; i++)
    a[i] = foo(i);
```

- deadlock: not all threads participating in a barrier

```
omp_set_num_threads(4);           $ export SUNW_MP_WARN=TRUE; ./a.out
#pragma omp parallel              At barrier 1.
{ int i = omp_get_thread_num();   At barrier 1.
  if (i % 2) {                     WARNING (libmstk): Threads at barrier from
    printf("At barrier 1.\n");     different directives.
    #pragma omp barrier            Thread at barrier from bar.c: 8.
                                  Thread at barrier from bar.c: 10.
  }
} /* implied barrier;line 10*/    ^c
```

## Review: Parallelizing Existing Applications for Multicore

- analyse the application for its parallelism potential
  - use a tool to do so, if available (e.g. SPARTAN, IWMSE'09)
  - study code base; estimate the refactoring necessary
- apply a systematic SDLC methodology (including for analysis & design, e.g. PADL)
  - will the design be scalable? how platform independent will it be?
- investigate tools that will assist parallel design and implementation
  - e.g. DPnDP: a GUI tool creating message-passing code from || design patterns
- attaining correctness:
  - build system in anticipation of difficult errors e.g. defensive programming, PBS checking
  - choose paradigms which reduce likelihood of errors (OpenMP)
  - use all available tools to detect or help find errors (esp. races & deadlocks!)
- attaining performance:
  - profiling tools are helpful, but need more (e.g. CoolTools' SPOT, SlowSpotter)
  - need for automated tools for tuning if parallelization is complex

## Thread Debugging Tools

- use any compiler and run-time checking available
- most debugging tools are threaded (e.g. gdb - Linux, dbx - Solaris)
  - e.g. (dbx) stop at 8 stops all thread in parallel region at line 8 (stops in 'outlined' (compiler-generated) function with name like \$d1A7.main)
  - can then use (dbx) thread t@1 to select thread, (dbx) where to examine stack, etc
- data race detection using Solaris Thread Analyser:

```
$ cc -g -xinstrument=datarace -xvpara -xopenmp -fast -o dsum_omp_atomic dsum_omp_atomic
"dsum_omp_atomic.c", line 30: Warning: inappropriate scoping
    variable 'sum' may be scoped inappropriately as 'shared'
    . write at line 33 and write at line 33 may cause data race
$ collect -r on -d /tmp/$USER ./dsum_omp_atomic 100000
Creating experiment database /tmp/u8914893/tha.1.er ..
$ analyzer /tmp/$USER/tha.1.er/ &
```